Galactic File System

Juan Benet
juan@benet.ai

ABSTRACT

The Galactic File System is a peer-to-peer distributed file system capable of sharing the same files with millions of nodes. GFS combines a distributed hashtable, cryptographic techniques, merkle trees, content-addressable storage, bittorrent, and tag-based filesystems to build a single massive file system shared between peers. GFS has no single point of failure, and nodes do not need to trust each other.

1. INTRODUCTION

[Motivate GFS. Introduce problems. Describe BitTorrent existing problems (multiple files. one swarm. sloppy dht implementation.) Describe version control efforts. Propose potential combinations of good ideas.]

[Cite: CFS, Kademlia, Bittorrent, Chord, DHash, SFS, Ori, Coral]

This paper introduces GFS, a novel peer-to-peer version-controlled filesystem; and BitSwap, the novel peer-to-peer block exchange protocol serving GFS.

The rest of the paper is organized as follows. Section 2 describes the design of the filesystem. Section 3 evaluates various facets of the system under benchmark and common workloads. Section 4 presents and evaluates a world-wide deployment of GFS. Section 5 describes existing and potential applications of GFS. Section 6 discusses related and future work.

Notation Notes: (a) data structures are specified in Go syntax, (b) rpc protocols are specified in capnp interface, (c) object formats are specified in text with <fields>.

2. DESIGN

2.1 GFS Nodes

GFS is a distributed file system where all nodes are the same. They are identified by a `NodeId`, the cryptographic hash of a public-key (note that *checksum* will henceforth refer specifically to crypographic hashes of an object). Nodes also store their public and private keys. Clients are free to instatiate a new node on every launch, though that means losing any accrued benefits. It is recommended that nodes remain the same.

```
type Checksum string
type PublicKey string
type PrivateKey string
type NodeId Checksum

type Node struct {
  nodeid NodeID
  pubkey PublicKey
  prikey PrivateKey
}
```

Together, the nodes store the GFS files in local storage, and send files to each other. GFS implements its features by combining several subsystems with many desirable properties:

1. A Coral-based **Distributed Sloppy Hash Table** (DSHT) to link and coordinate peer-to-peer nodes. Described in Section 2.2.
2. A Bittorrent-like peer-to-peer **Block Exchange** (BE) distribute Blocks efficiently, and to incentivize replication. Described in Section 2.3.
3. A Git-inspired **Object Model** (OM) to represent the filesystem. Described in Section 2.4.
4. An SFS-based self-certifying name system. Described in Section 2.5.

These subsystems are not independent. They are well integrated and leverage their blended properties. However, it is useful to describe them separately, building the system from the bottom up. Note that all GFS nodes are identical, and run the same program.

2.2 Distributed Sloppy Hash Table

First, GFS nodes implement a DSHT based on Kademlia and Coral to coordinate and identify which nodes can serve a particular block of data.

2.2.1 Kademlia DHT

Kademlia is a DHT that provides:

1. Efficient lookup through massive networks: queries on average contact $\lceil log_2(n) \rceil$ nodes. (e.g. 20 hops for a network of 10000000 nodes).

2. Low coordination overhead: it optimizes the number of control messages it sends to other nodes.

3. Resistance to various attacks, by preferring nodes who have been part of the DHT longer.

4. wide useage in peer-to-peer applications, including Gnutella and Bittorrent, forming networks of over 100 million nodes.

While some peer-to-peer filesystems store data blocks directly in DHTs, this "wastes storage and bandwidth, as data must be stored at nodes where it is not needed". Instead, GFS stores a list of peers that can provide the data block.

2.2.2 Coral DSHT

Coral extends Kademlia in three particularly important ways:

1. Kademlia stores values in nodes whose ids are "nearest" (using XOR-distance) to the key. This does not take into account application data locality, ignores "far" nodes who may already have the data, and forces "nearest" nodes to store it, whether they need it or not. This wastes significant storage and bandwith. Instead, Coral stores addresses to peers who can provide the data blocks.

2. Coral relaxes the DHT API from `get_value(key)` to `get_any_values(key)` (the "sloppy" in DSHT). This still works since Coral users only need a single (working) peer, not the complete list. In return, Coral can distribute only subsets of the values to the "nearest" nodes, avoiding hot-spots (overloading *all the nearest nodes* when a key becomes popular).

3. Additionally, Coral organizes a hierarchy of separate DSHTs called *clusters* depending on region and size. This enables nodes to query peers in their region first, "finding nearby data without querying distant nodes" and greatly reducing the latency of lookups.

2.2.3 GFS DSHT

The GFS DSHT supports four RPC calls:

2.3 Block Exchange - BitSwap Protocol

The exchange of data in GFS happens by exchanging blocks with peers using a BitTorrent inspired protocol: BitSwap. Like BitTorrent, BitSwap peers are looking to acquire a set of blocks, and have blocks to offer in exchange. Unlike BitTorrent, BitSwap is not limited to the blocks in one torrent. BitSwap operates as a persistent marketplace where node can acquire the blocks they need, regardless of what files the blocks are part of. The blocks could come from completely unrelated files in the filesystem. But nodes come together to barter in the marketplace.

While the notion of a barter system implies a virtual currency could be created, this would require a global ledger (blockchain) to track ownership and transfer of the currency. This will be explored in a future paper.

Instead, BitSwap nodes have to provide direct value to each other in the form of blocks. This works fine when the distribution of blocks across nodes is such that they have the complements, what each other wants. This will seldom be the case. Instead, it is more likely that nodes must *work* for their blocks. In the case that a node has nothing that its peers want (or nothing at all), it seeks the pieces its peers might want, with lower priority. This incentivizes nodes to cache and disseminate rare pieces, even if they are not interested in them directly.

2.3.1 BitSwap Credit

The protocol must also incentivize nodes to seed when they do not need anything in particular, as they might have the blocks others want. Thus, BitSwap nodes send blocks to their peers, optimistically expecting the debt to be repaid. But, leeches (free-loading nodes that never share) must be avoided. A simple credit-like system solves the problem:

1. Peers track their balance (in bytes verified) with other nodes.

2. Peers send blocks to debtor peers probabilistically, according to a function that falls as debt increases.

Note that if a peer decides not to send, the peer subsequently ignores the other node for an `ignore_cooldown` timeout. This prevents senders from trying to game the probability by just causing more dice-rolls. (Default BitSwap is 10 seconds).

2.3.2 BitSwap Strategy

The differing strategies that BitSwap peers might employ have wildly different effects on the performance of the exchange as a whole. In BitTorrent, while a standard strategy is specified (tit-for-tat), a variety of others have been implemented, ranging from BitTyrant (sharing the least-possible), to BitThief (exploiting a vulnerability and never share), to PropShare (sharing proportionally). A range of strategies (good and malicious) could similarly be implemented by BitSwap peers. The choice of function, then, should aim to:

1. maximize the trade performance for the node, and the whole exchange

2. prevent freeloaders from exploiting and degrading the exchange

3. be effective with and resistant to other, unknown strategies

4. be lenient to trusted peers

The exploration of the space of such strategies is future work. One choice of function that works in practice is the sigmoid, scaled by a *debt retio*:

Let the *debt ratio* r between a node and its peer be:

$$r = \frac{\texttt{bytes\_sent}}{\texttt{bytes\_recv} + 1}$$

Given r, let the probability of sending to a debtor be:

$$P\Big(\ send \mid r\ \Big) = 1 - \frac{1}{1 + exp(6 - 3r)}$$

As you can see in Figure **??**, this function drops off quickly as the nodes' *debt ratio* surpasses twice the established credit. The *debt ratio* is a measure of trust: lenient to debts between nodes that have previously exchanged lots of data successfully, and merciless to unknown, untrusted

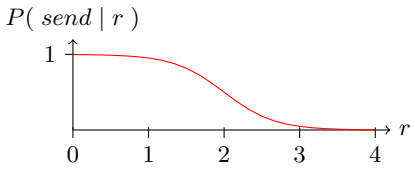

Figure 1: Probability of Sending as r increases

nodes. This (a) provides resistance to attackers who would create lots of new nodes (sybill attacks), (b) protects previously successful trade relationships, even if one of the nodes is temporarily unable to provide value, and (c) eventually chokes relationships that have deteriorated until they improve.

2.3.3 BitSwap Ledger

BitSwap nodes keep ledgers accounting the transfers with other nodes. A ledger snapshot contains a pointer to the previous snapshot (its checksum), forming a hash-chain. This allows nodes to keep track of history, and to avoid tampering. When activating a connection, BitSwap nodes exchange their ledger information. If it does not match exactly, the ledger is reinitialized from scratch, loosing the accrued credit or debt. It is possible for malicious nodes to purposefully "loose" the Ledger, hoping the erase debts. It is unlikely that nodes will have accrued enough debt to warrant also losing the accrued trust, however the partner node is free to count it as *misconduct* (discussed later).

```
type Ledger struct {
  parent     Checksum
  owner      NodeId
  partner    NodeId
  bytes_sent int
  bytes_recv int
  timestamp  Timestamp
}
```

Nodes are free to keep the ledger history, though it is not necessary for correct operation. Only the current ledger entries are useful. Nodes are also free to garbage collect ledgers as necessary, starting with the less useful ledgers: the old (peers may not exist anymore) and small.

2.3.4 BitSwap Specification

BitSwap nodes follow a simple protocol.

```
# Additional state kept:
type BitSwap struct {
  ledgers map[NodeId]Ledger
  // Ledgers known to this node, inc inactive

  active map[NodeId]Peer
  // currently open connections to other nodes

  need_list []Checksum
  // checksums of blocks this node needs

  have_list []Checksum
  // checksums of blocks this node has
}
```

```
type Peer struct {
  nodeid NodeId
  ledger Ledger
  // Ledger between the node and this peer

  last_seen Timestamp
  // timestamp of last received message

  want_list []Checksum
  // checksums of all blocks wanted by peer
  // includes blocks wanted by peer's peers
}

# Protocol interface:
interface Peer {
  open (nodeid :NodeId, ledger :Ledger);
  send_want_list (want_list :WantList);
  send_block (block :Block) -> (complete :Bool);
  close (final :Bool);
}
```

Sketch of the lifetime of a peer connection:

1. Open: peers send `ledgers` until they agree.
2. Sending: peers exchange `want_lists` and `blocks`.
3. Close: peers deactivate a connection.
4. Ignored: (special) a peer is ignored (for the duration of a timeout) if a node's strategy avoids sending

Peer.open(NodeId, Ledger).

When connecting, a node initializes a connection with a `Ledger`, either stored from a connection in the past or a new one zeroed out. Then, sends an Open message with the `Ledger` to the peer.

Upon receiving an `Open` message, a peer chooses whether to activate the connection. If – acording to the receiver's `Ledger` – the sender is not a trusted agent (transmission below zero, or large outstanding debt) the receiver may opt to ignore the request. This should be done probabilistically with an `ignore_cooldown` timeout, as to allow errors to be corrected and attackers to be thwarted.

If activating the connection, the receiver initializes a Peer object, with the local version of the `Ledger`, and setting the `last_seen` timestamp). Then, it compares the received `Ledger` with its own. If they match exactly, the connections have opened. If they do not match, the peer creates a new zeroed out `Ledger`, and sends it.

Peer.send_want_list(WantList).

While the connection is open, nodes advertise their `want_list` to all connected peers. This is done (a) upon opening the connection, (b) after a randomized periodic timeout, (c) after a change in the `want_list` and (d) after receiving a new block.

Upon receiving a `want_list`, a node stores it. Then, it checks whether it has any of the wanted blocks. If so, it sends them according to the *BitSwap Strategy* above.

Peer.send_block(Block).

Sending a block is straightforward. The node simply transmits the block of data. Upon receiving all the data, the

receiver computes the Checksum to verify it matches the expected one, and returns confirmation.

Upon finalizing the correct transmission of a block, the receiver moves the block from `need_list` to `have_list`, and both the receiver and sender update their ledgers to reflect the additional bytes transmitted.

If a transmission verfication fails, the receiver instead *penalizes* the sender. Both receiver and sender should update their ledgers accordingly, though the sender is either malfunctioning or attacking the receiver. Note that BitSwap expects to operate on a reliable transmission channel, so data errors – which could lead to incorrect penalization of an honest sender – are expected to be caught before the data is given to BitSwap. GFS uses the uTP protocol.

Peer.close(Bool).

The `final` parameter to `close` signals whether the intention to tear down the connection is the sender's or not. If false, the receiver may opt to re-open the connection immediatelty. This avoids premature closes.

A peer connection should be closed under two conditions:

- a `silence_wait` timeout has expired without receiving any messages from the peer (default BitSwap uses 30 seconds). In this case, the node issues a `Peer.close(false)` message.
- the node is exiting and BitSwap is being shut down. In this case, the node issues a `Peer.close(true)` message.

After a `close` message, both receiver and sender tear down the connection, clearing any state stored. The `Ledger` may be stored for the future, if it is useful to do so.

Notes.

- Non-`open` messages on an inactive connection should be ignored. In case of a `send_block` message, the receiver may check the block to see if it is needed and correct, and if so, use it. Regardless, all such out-of-order messages trigger a `close(false)` message from the receiver, to force re-initialization of the connection.

2.4 Object Model

The DHT and BitSwap allow GFS to form a massive peer-to-peer system for storing and distributing blocks quickly and robustly to users. GFS builds a filesystem out of this efficient block distribution system, constructing files and directories out of blocks.

Files in GFS are represented as a collection of inter-related objects, like in the version control system Git. Each object is addressed by the cryptographic hash of its contents (`Checksum`). The file objects are:

1. `block`: a variable-size block of data.
2. `list`: a collection of blocks or other lists.
3. `tree`: a collection of blocks, lists, or other trees.
4. `commit`: a snapshot in the version history of a tree.

We hoped to use the Git object formats exactly, but had to depart to introduce certain features useful in a distributed filesystem, for example fast size lookups (aggregate byte sizes have been added to objects), large file deduplication and versioning (adding a `list` object), and more. However, our objects are perfectly compatible with Git and conversion between the two does not lose any information.

Notes:

- `varint` is a variable size int, as in protocol-buffers.
- objects are serialized using `capnp`.

2.4.1 Block Object

The `Block` object contains an addressable unit of data, and represents a file. GFS Blocks are like Git blobs or filesystem data blocks. They store the users' data. (The name *block* is preferred over *blob*, as the Git-inspired view of a *blob* as a *file* breaks down in GFS. GFS files can be represented by both `lists` and `blocks`.) Format:

```
block <size>
<block data bytes>
...
```

2.4.2 List Object

The `List` object represents a large or de-duplicated file made up of several GFS `Blocks` concatenated together. `Lists` contain an ordered sequence of `block` or `list` objects. In a sense, the GFS `List` functions like a filesystem file with indirect blocks. Since `lists` can contain other `lists`, topologies including linked lists and balanced trees are possible. Directed graphs where the same node appears in multiple places allow in-file deduplication. Cycles are not possible (enforced by hash addessing). Format:

```
list <num objects> <size varint>
<list or block> <checksum> <size varint>
<list or block> <checksum> <size varint>
...
```

2.4.3 Tree Object

The `tree` object in GFS is similar to Git trees: it represents a directory, a list of checksums and names. The checksums reference `blob` or other `tree` objects. Note that traditional path naming is implemented entirely by the `tree` objects. `Blocks` and `lists` are only addressed by their `checksums`. Format:

```
tree <num objects> <size varint>
<tree or list or block> <checksum> <size varint> <name>
<tree or list or block> <checksum> <size varint> <name>
...
```

2.4.4 Commit Object

The `commit` object in GFS is similar to Git's. It represents a snapshot in the version history of a `tree`. Note that user addresses are NodeIds (the hash of the public key).

```
commit <size varint>
parent <commit checksum>
tree <tree checksum>
author <author public key> <ISO UTC date>
committer <committer public key> <ISO UTC date>
<commit message>
```

2.4.5 *Version control*

The `commit` object represents a particular snapshot in the version history of a tree. Comparing the `trees` and children objects of two different commits reveals the differences between two versions of the filesystem. As long as a single `commit` and all the children objects it references are accessible, all preceding versions are retrievable and the full history of the filesystem changes can be accessed. This is a consequence of the `Git` object model and the graph it forms.

The full power of the `Git` version control tools is available to GFS users. The object model is compatible (though not the same). The standard `Git` tools can be used on the `GFS` object graph after a conversion. Additionally, a fork of the tools is under development that will allow users to use them directly without conversion.

2.4.6 *Object-level Cryptoraphy*

GFS is equipped to handle object-level cryptographic operations. Any additional bytes are appended to the bottom of the object. This changes the object's hash (defining a different object, as it should). GFS exposes an API that automatically verifies signatures or decrypts data.

- `Signing`. Signature appended.
- `Encryption`. Optional recipient's public key appended.

2.4.7 *Merkle Trees*

The object model in GFS forms a *Merkle Tree*, which provides GFS with useful properties:

1. **Content Addressing:** all content is uniquely identified by its `checksum`, **including child checksums**. This means a particular `tree` object points to *specific* children. Committing changes to a `block` also commits changes to the containing `tree`.
2. **Tamper resistance:** all content is verified with its Checksum. If data is tampered with, before being delivered, the client detects and discards it.
3. **Deduplication:** all objects who hold the exact same content are the same, and only stored once. This is particularly useful with parent objects, such as lists, trees, and commits.

2.5 The Filesystem

2.5.1 *Filesystem Paths*

GFS exposes a slash-delimited path-based API. Paths work the same as in any traditional UNIX filesystem. Path subcomponents have different meanings per object:

`object`	subcomponent meaning
`block`	N/A (no children)
`list`	integer index
`tree`	string name
`commit`	string name (in tree)

For example, given the sample objects in Figures **??** and **??**:

```
# to access tree ttt333:
ccc111/ttt333-name
```

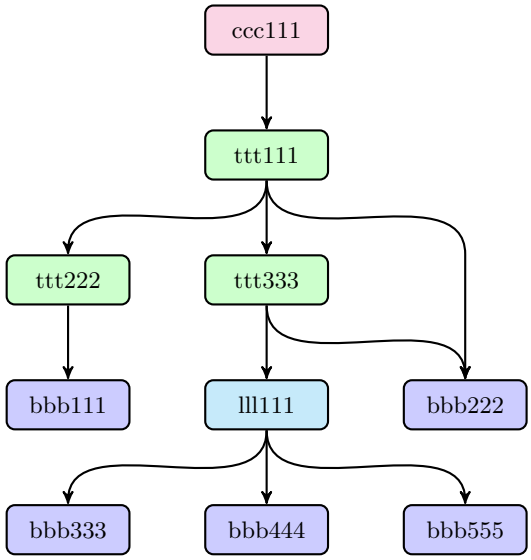

Figure 2: Sample Object Graph

```
# ccc111 contents
commit 313
tree ttt111
author <author public key> <ISO UTC date>
committer <committer public key> <ISO UTC date>

# ttt111 contents
tree 3 250
tree ttt222 46 ttt222-name
tree ttt333 166 ttt333-name
block bbb222 11 bbb222-name

# ttt222 contents
tree 1 10
block bbb111 10 bbb111-name

# ttt333 contents
tree 2 104
list lll111 93 lll111-name
block bbb222 11 bbb222-eman

# lll111 contents
list 3 39
block bbb333 12
block bbb444 13
block bbb555 14

# bbb111 contents      # block bbb222 contents
block 1                block 2
1                      22

# bbb333 contents      # block bbb444 contents
block 3                block 4
333                    4444

# bbb555 contents
block 5
55555
```

Figure 3: Sample Objects

```
tree 5 250
tree ttt222 46 ttt222-name
block bbb111 10 ttt222-name/bbb111-name
tree ttt333 166 ttt333-name
list lll111 93 ttt222-name/lll111-name
block bbb222 11 ttt333-name/bbb222-eman
block bbb222 11 bbb222-name
```

Figure 4: Flattened Tree for `ttt111`

```
# to access block bbb222:
ccc111/bbb222-name
ccc111/ttt333-name/bbb222-eman

# to access list lll111:
ccc111/ttt333-name/lll111-name

# to access block bbb555:
ccc111/ttt333-name/lll111-name/2
```

Note that:

(a) blocks have no children
`.../<block>/<child>` is impossible

(b) commits implicitly access their trees
`.../<commit>/name` looks up `"name"` in `<commit>`'s `<tree>`

(c) `list` children are accessed by their index
`.../<list>/4` looks up the fifth block.

Path Lookup Performance.
Path-based access traverses the object graph. Retrieving each object requires potentially looking up its key in the DHT, connecting to peers, and retrieving its blocks. This is considerable overhead, particularly when looking up paths with many components. This is mitigated by:

- **tree caching**: since all objects are hash-addressed, they can be cached indefinitely. Additionally, `trees` tend to be small in size so GFS prioritizes caching them over `blocks`.
- **flattened trees**: for any given `tree`, a special `flattened tree` can be constructed to list all objects reachable from the `tree`. Figure **??** shows an example of a flattened tree. While GFS does not construct flattened trees by default, it provides a function for users. For example,

2.5.2 Publishing Objects

GFS is globally distributed. It is designed to allow the files of millions of users to coexist together. The **DHT** with content-hash addressing allows publishing objects in a fair, secure, and entirely distributed way. Anyone can publish an object by simply adding its key to the DHT, adding themselves as a peer, and giving other users the object's hash.

Additionally, the GFS root directory supports special functionality to allow namespacing and naming objects in a fair, secure, and distributed manner.

(a) All objects are accessible by their hash. Thus, users can always reference an object (and its children) using `/<object_hash>`.

(b) `/<node_id>` provides a self-certifying filesystem for user `node_id`. If it exists, the object returned is a special `tree` signed by `node_id`'s private key. Thus, a user can publish a `tree` or `commit` under their name, and others can verify it by checking the signature matches.

(c) If `/<domain>` is a valid domain name, GFS looks up key `gfs` in its `DNS TXT` record. GFS interprets the value as either an object hash or another GFS path:

```
# this DNS TXT record
fs.benet.ai. TXT "gfs=/aabbccddeeffgg ..."

# behaves as symlink
ln -s /aabbccddeeffgg /fs.benet.ai
```

2.6 Local Objects

GFS clients require some *local storage*, an external system on which to store and retrieve local raw data for the objects GFS manages. The type of storage depends on the node's use case. In most cases, this is simply a portion of disk space (either managed by the native filesystem, or directly by the GFS client). In others, non- persistent caches for example, this storage is just a portion of RAM.

Ultimately, all blocks available in GFS are in some node's *local storage*. And when nodes open files with GFS, the objects are downloaded and stored locally, at least temporarily. This provides fast lookup for some configurable amount of time thereafter.

2.6.1 Object Pinning

Nodes who wish to ensure the survival of particular objects can do so by `pinning` the objects. This ensures the objects are kept in the node's *local storage*. Pinning can be done recursively, to pin down all descendent objects as well. For example, recursively pinning a `tree` or `commit` ensures *all* objects pointed to are stored locally too. This is particularly useful for nodes wishing to keep all their own files.